(4)

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: NW-LIS-89-12-05	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): Timing Optimization of Multi-Phase Sequential Logic	5. TYPE OF REPORT & PERIOD COVERED: Technical
6. PERFORMING ORG. REPORT NUMBER	
7. AUTHOR(s): Karen Bartlett, Gaetano Borriello, Sitaram Raju	8. CONTRACT OR GRANT NUMBER(s): N00014-88-K-0453
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Northwest Laboratory for Integrated Systems, University of Washington, Dept. of Comp. Science, FR-35 Seattle, WA 9819[illegible]	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: DARPA-ISTO, 1400 Wilson Boulevard, Arlington, VA 22209	12. REPORT DATE: December 1989
	13. NUMBER OF PAGES: 12
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Office of Naval Research - ONR, Information Systems Program - Code 1513: CAF, 800 North Quincy Street, Arlington, VA 22217	15. SECURITY CLASS. (of this report): Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this report is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Timing optimization, multi-phase logic, sequential logic, logic optimization, multi-phase clock, MISII, area optimization, retiming, VLSI circuits.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

High-performance MOS circuits are frequently designed using pre-charged and dynamic logic. This requires the use of multiple phases of the system clock to ensure that the circuitry is pre-charged and refreshed at the proper times during each clock cycle. Finite-state machines used to control this type of logic must therefore be constructed as multi-phase sequential logic with inputs

Continued on back...

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 68 IS OBSOLETE S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

#20 ABSTRACT

Continued from front page.

and outputs stable during the appropriate phases. The timing optimization of multi-phase logic entails the reduction of the overall cycle time of the machine as well as input to output delays by distributing computation throughout the entire clock cycle. Currently, no tools are available to automatically perform this optimization task for multi-phase logic. We have developed such a tool as a set of extensions to the combinational logic optimization tool, misII. Our algorithms yield improvements that are 20% better than what is achievable using only combinational logic optimization tools that do not move logic across latches. Furthermore, we achieve 65% of the improvements possible in the most idealized case. Results on simple two-phase circuits show average input to output delay improvements of almost 20% with area penalties of less than 12%. For a four-phase controller used in the SPUR processor it yields an improvement in cycle time of 21% with an area penalty of 21%.

Timing Optimization of Multi-Phase Sequential Logic

Karen Bartlett, Gaetano Borriello, Sitaram Raju

Technical Report #89-12-05

Department of Computer Science and Engineering, FR-35
University of Washington, Seattle, WA 98195 USA

DEPARTMENT OF COMPUTER SCIENCE
University of Washington
Seattle 98195

Timing Optimization of Multi-Phase Sequential Logic

Karen Bartlett, Gaetano Borriello, Sitaram Raju

Technical Report #89-12-05

Department of Computer Science and Engineering, FR-35
University of Washington, Seattle, WA 98195 USA

Timing Optimization of Multi-Phase Sequential Logic

Karen Bartlett, Gaetano Borriello, Sitaram Raju

Department of Computer Science and Engineering, FR-35
University of Washington
Seattle, WA 98195

Abstract

High-performance MOS circuits are frequently designed using pre-charged and dynamic logic. This requires the use of multiple phases of the system clock to ensure that the circuitry is pre-charged and refreshed at the proper times during each clock cycle. Finite-state machines used to control this type of logic must therefore be constructed as multi-phase sequential logic with inputs and outputs stable during the appropriate phases. The timing optimization of multi-phase logic entails the reduction of the overall cycle time of the machine as well as input to output delays by distributing computation throughout the entire clock cycle. Currently, no tools are available to automatically perform this optimization task for multi-phase logic. We have developed such a tool as a set of extensions to the combinational logic optimization tool, `misII`. Our algorithms yield improvements that are 20% better than what is achievable using only combinational logic optimization tools that do not move logic across latches. Furthermore, we achieve 65% of the improvements possible in the most idealized case. Results on simple two-phase circuits show average input to output delay improvements of almost 20% with area penalties of less than 12%. For a four-phase controller used in the SPUR processor it yields an improvement in cycle time of 21% with an area penalty of 21%.

I. Introduction

The design of digital MOS circuits is simplest and most robust when fully complementary combinational logic and edge-triggered static storage elements are used. This style of design permits the circuit to be controlled by a single-phase clock and eliminates concerns about dynamic storage times. However, this type of design is usually not the densest nor the fastest possible.

The design of high-density high-performance MOS circuits usually requires the use of pre-charged combinational logic and level-sensitive dynamic storage elements. Pre-charged logic eliminates approximately one half of the transistors required to implement a logic function (yielding higher density) and can eliminate long chains of transistors connected in series (yielding higher performance).

The price that must be paid is that, unlike static design, these types of circuits require the careful management of computation flow during the clock cycle to ensure correct operation. For example, while the output of fully complementary combinational logic can be sampled throughout the clock cycle, the output of a circuit that utilizes pre-charging can only be sampled while it is not being precharged. This naturally leads to at least two phases of a clock cycle: one used for pre-charging and one used for evaluation. Also, while edge-triggered static latches can be controlled by a single-phase clock, level-sensitive latches require at least two phase clocks to control the flow of data around feedback paths.

The general class of circuits built using these more complex and higher performance timing methodologies is referred to as *multi-phase sequential logic*. Multi-phase finite-state machines have inputs and outputs that are only valid during a subset of the clock period corresponding to one or more phases of the clock. For example, while in single-phase edge-triggered designs the output of one block can be an input to any other block, in multi-phase designs the phases during which a signal is stable determine how it can be used. In some cases, a signal may need to be delayed and synchronized to another phase of the clock before it can be used as an input to another logic block.

In designing multi-phase sequential logic, the designer must determine the phase during which each logic function will be performed. There are two extreme cases. Outputs dependent on inputs from the immediately previous phase restrict the placement of their logic to a single phase. The other extreme is the state logic of a finite-state machine which can be spread over all the phases of the clock cycle. In general, it is difficult to determine which placement will lead to the

overall smallest or fastest design. Furthermore, timing optimization of this class of circuits has two aspects: the minimization of both the overall cycle time and the delay from primary inputs to primary outputs.

A Model of Multi-Phase Sequential Logic

Figure 1 shows three phases of a clock and a graph for the structure of the multi-phase network used for this timing scheme. In an n-phase system the individual phases are labeled ϕ_1 through ϕ_n where the occurrence of ϕ_n in one cycle immediately precedes the occurrence of ϕ_1 in the next. A node i represents the latches that are active during ϕ_i. An edge (i,j) represents the block of combinational logic (C_{ij}) that generates the signals output on ϕ_j (where ϕ_i immediately precedes ϕ_j). There are n logic blocks, one for each phase of the clock. The time available for computation in block C_{ij} is measured from the rising edge of ϕ_i (the block's *start phase*) to the rising edge of ϕ_j (the block's *end phase*).

The logic of the original specification is partitioned into blocks. Primary output and internal state logic is placed in block whose end phase is the phase during which the output signal is latched. When signals are dependent on an input that arrives earlier than the start phase of the block then that input is delayed through the earlier block. Therefore some inputs will simply be passed through one block to another. Signals output more than once during a cycle will have their logic duplicated. For example, a gate that has one input sampled on ϕ_1 and another on ϕ_3 and whose output is required during both ϕ_2 and ϕ_4 will be duplicated in blocks C_{12} and C_{34} and all four blocks will pass at least one input to later blocks.

The cycle time of the sequential logic is defined as the sum of the delays along the cycle in the graph passing through all the logic blocks and latches. Latches are required for every input and output to sample and hold the data values. However, they are not required at every connection between the blocks as the graph model may at first seem to imply. Latches are only needed to break cycles in the logic and ensure that no race conditions exist through the latches. If edge-triggered latches are used then every cycle must be broken by at least one latch. In the case of level sensitive latches, at least two latches clocked by different phases of the clock are required and the two phases must underlap (i.e., both are inactive) at some point during the clock cycle.

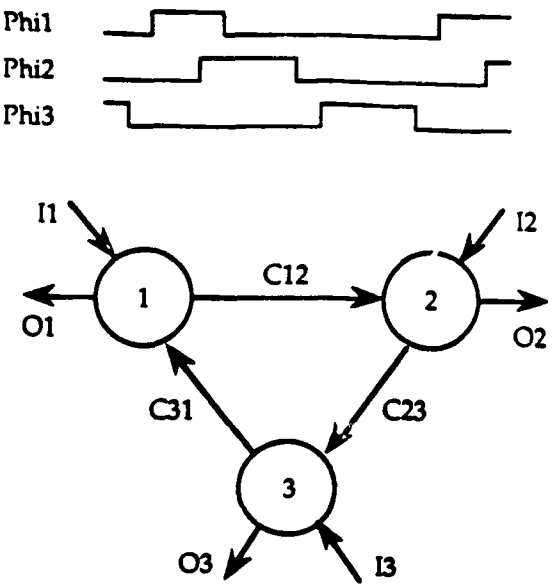

Figure 1

The longest path through a block and the propagation delay of any latches on the inputs of this path determine the time between the rising edges of the start and end phases of the block (i.e., the time from when the input latches of the block are loaded with new data and it propagates through the latch and logic of the block and appears at the output). These computation times will be used to precisely position the rising edges of each clock phase within the clock cycle used to control the circuit. The duration of each of the phases (i.e., the time between the rising and falling edge) is bounded from below by the minimum time required for the latch to be active and properly store the input value and from above by the requirement of preventing race conditions along the feedback paths in the circuit. That is, if all the latches along a cycle in the circuit are active then the fastest logic along that cycle must be slower than the overlap time of the phases along the path. This restriction is both necessary and sufficient to ensure proper operation. Many designs adopt a non-overlapped clocking scheme precisely to avoid this added constraint. In that case, the position of the falling edge is determined by the position of the rising edge of the next phase. This is not an issue with edge-triggered latches where the upper bound on the duration of a phase is simply the clock period.

In some cases, inputs may arrive earlier than the rising edge of the start phase that will sample their value into a latch. This *early arrival time*

permits the movement of some combinational logic before the latch and thereby reduces the amount of computation once the new values enter the block. Logic pre-computed in this manner is termed *pre-block logic* (*pre-C_{ij}*). Similarly, outputs may not be required for some time after the rising edge of the end phase that will load their value into a latch. This *late required time* permits the movement of some combinational logic after the latch and thereby reduces the amount of computation required to produce the output before it leaves the block. Logic post-computed in this manner is termed *post-block logic* (*post-C_{ij}*).

Related Optimization Approaches

Currently, there are no automatic tools to help designers with the timing optimization of multi-phase sequential logic. However, there are two types of tools that might serve as a starting point for automating this process. The first approach uses retiming algorithms while the second exploits combinational logic optimization tools but neither approach alone nor a combination of the two is enough to address the problem for multi-phase logic.

Global retiming algorithms such as the algorithm of Leiserson and Saxe [LEIS83, SAXE85] can optimally reposition latches within a Boolean network so as to minimize overall cycle time. However, the linear programming formulation used to solve this optimization problem cannot be easily and efficiently extended to consider restructuring of the Boolean logic. Often, minor changes in logic permit the realization of a much faster circuit. Another deficiency of this approach is that it assumes a single-phase clock. This restriction can be relaxed somewhat to address multi-phase logic if each phase is viewed as an entire clock cycle and therefore implicitly restricted to being of the the same duration and appropriate constraint equations added to the linear program formulation. However, this is impractical in high-performance designs where each phase is compressed as much as possible. Also, there is no method for removing unneeded latches (see Section 4) except in a post-processing step where timing information cannot be used to optimize the placement of the remaining latches.

Logic optimization tools such as `misII` [BRAY87, SING88] and `Bold` [BART87] can dramatically improve the area and performance of combinational circuits. However, these optimizations only operate on the distinct combinational blocks defined by the initial latch boundaries. Recent algorithms [DEMI89, MALI89] have utilized both logic optimization and global retiming techniques. However, these approaches are still fundamentally limited by the global retiming algorithms to single-phase or equal duration multi-phase logic.

Structure of the Paper

The paper is divided into six main sections. The first section is this introduction to the problem domain and the terminology required to discuss multi-phase sequential logic. Section 2 introduces the basic transformations that can be used to restructure a circuit and improve its timing characteristics. Section 3 presents an overview of the algorithm we have implemented to automate the timing optimization process. Section 4 explains the required latch manipulations and optimizations. Section 5 describes the details of the extensions to the logic and timing optimization routines provided by `misII`. Section 6 details the results obtained by our algorithm on a collection of examples including a subset from the International Logic Synthesis Workshop benchmark set [LISA88] and a four-phase controller from the SPUR CPU design [HILL88, KONG89]. Section 7 concludes the paper with some summary remarks and a description of future work.

II. Basic Transformations

In this section, we use a small two-phase sequential circuit to outline the basic transformations necessary for timing optimization. These transformation involve the movement of combinational logic across latch boundaries.

A Brief Example

Figure 2 shows the structure of a simple two-phase finite-state machine that was designed as part of a custom routing chip for a parallel processor interconnection network at the University of Washington [EBEL88]. It has inputs and outputs on both phases and two blocks of combinational logic (the circles represent latches).

The first problem encountered in the design of this machine is in its specification. There are at least three equivalent ways to describe the machine. They differ only on when state transitions occur during a clock cycle. First, all the transitions can be on ϕ_1 so that all the inputs arriving on the previous ϕ_1 must be delayed until the next ϕ_2 and a state

diagram constructed based on that model. Second, the transitions can be on ϕ_2 and the inputs on ϕ_2 must be delayed to the next ϕ_1. And third, transitions between states can be permitted on both phases of the clock.

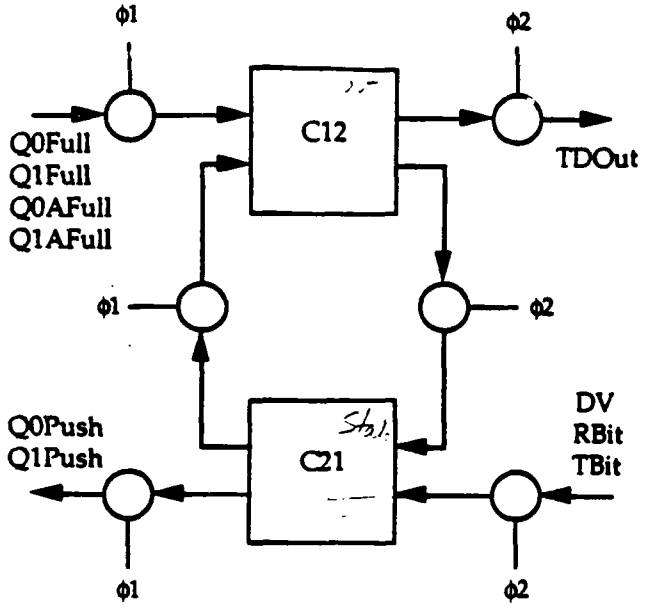

Figure 2

Finite-state machine synthesis tools currently support only the first two models with the translation of inputs (and similarly the outputs) performed by the designer before specification. These models yield two machines described in Table 1 entries *most-C12* and *most-C21*. Most of the logic is in one of the two blocks with only enough in the other block to ensure that no extra phase translation delays are introduced on input to output paths. The two machines differ substantially in terms of the number of latches, logic literal count, and cycle time. However, given that the two synthesizable machines are input-output equivalent, it should be possible to implement a synthesis tool that can transform one machine into the other and reach all the intermediate implementations as well. One such intermediate representation is the one represented by the third specification option and the third row in Table 1, *C12-C21*. The three machines all have the same basic structure (see Figure 2) but with some important differences. In each, the C_{12} logic block computes a function of the inputs and state sampled on ϕ_1 (Q0Full, Q1Full, Q1AFull, Q0AFull) and generates an output on ϕ_2 (TDout). Similarly, the C_{21} logic block computes a function of inputs and state sampled on ϕ_2 (DV, TBit, RBit) and generates outputs on ϕ_1 (Q0Push, Q1Push).

	C12 del	C12 lit	C21 del	C21 lit	latches	tot area	cycle time
most-C12	4.2	14	7.5	72	20	126	11.7
most-C21	9.5	72	4.8	16	25	138	14.3
C12-C21	4.7	29	6.7	49	19	116	11.4

Table 1

The three implementations differ in where the next-state computations are performed. In *most-C21*, the state computation is performed entirely in the C_{21} block. The state bits are inputs to the C_{12} block and are also forwarded as inputs to C_{21} after being delayed one phase so as to be synchronized with the other inputs to C_{21}. Block C_{12} also forwards the value of its $\phi 1$ inputs to C_{21} after they are synchronized to $\phi 2$. In the symmetric implementation, *most-C12*, the state computation is performed entirely in C_{12}. C_{21} forwards both state bits and ϕ_2 inputs to C_{12}. The *C12-C21* implementation has state computation occurring in both blocks with the state bits changing on every phase rather than every cycle as in the other two implementations. This is what makes the specification of this implementation difficult for current finite-state machine synthesis tools. Furthermore, the logic of this implementation is more balanced across the two blocks, it is smaller in both literals and latches (latches are counted as 2 literals, i.e., a 2-input NAND), and is the fastest of the three.

All three implementations of the state machine are input-output equivalent but have different state bits (defined as the values stored in the dynamic latches controlled by the clock phases). In fact, they can be viewed as being derived from each other by a series of transformations that move combinational logic across latch boundaries. Many other implementations are possible besides those shown of Table 1. Our objective is to find one that has the best timing characteristics in terms of overall cycle time as well as input-output delay. The remainder of this section is a more detailed discussion of how logic is moved between logic blocks allowing us to move through the design space and find better implementations with respect to timing.

Moving Logic Forward and Backward

Logic at the head of a logic block (i.e., logic with fanout only going to an output of the block) can be moved out of the block, across any latches on the output, and into the tail of a subsequent logic block. Similarly, logic at the tail of a logic block (i.e., logic with fanin coming directly from an input of the block) can be moved out of the block, across any latches on the inputs, and into a previous logic block. These operations are illustrated in Figure 3.

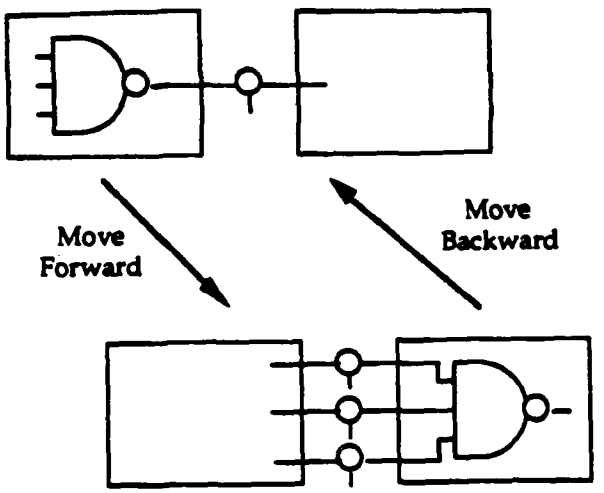

Figure 3

The effect of the moves is two-fold. First, the number of output or input latches is changed and therefore a different set of logic values will now be stored as state changing the state assignment for the finite-state machine. Second, the critical path delay of the two logic blocks may also be changed. If the logic to be moved is on a critical path of a block then the duration of the clock phase for that logic block may be shortened. However, if the logic is moved onto a critical path in the other logic block it may lengthen the duration of the clock phase for that block.

In cases where no logic can be moved with the existing configuration of logic gates restructuring may be required to generate logic that can be moved. Restructuring does not change the logic function but only its decomposition into logic gates. Examples of restructuring include moving late arriving inputs later in the network (as in `misII` speed-up [SING88]) or simply decreasing the grain size of the logic by decomposing a larger gate into smaller ones.

These transformations form the basis for our algorithm for timing optimization. Our approach is to select the appropriate blocks of logic to be moved either forward or backward to other blocks so as to minimize overall cycle time. Initially, the combinational logic of each block is optimized for timing and area using `misII`.

The *C12-C21* implementation of Table 1 was derived from an initial implementation (*most-C21*) obtained by using standard finite-state machine synthesis tools. The output of the tools was modified by placing some logic in C_{12} so as to not incur extra input to output delays due to phase synchronization. In deriving *C12-C21*, logic was moved back from C_{21} to C_{12}. The logic that is moved depends only on inputs that are valid on ϕ_1 and thus can be computed earlier (i.e., moved back to C_{12}), reducing the compute time of C_{21}. If as much logic as possible is moved from C_{21} to C_{12} then we arrive at the *most-C12* implementation.

III. The Timing Optimization Algorithm

Our algorithm for the timing optimization of multi-phase logic is similar in approach to the logic optimization algorithm of `misII`. A collection of routines have been defined that implement the sequential logic transformations and restructuring outlined in the previous section. The timing optimization algorithm is a script that uses these routines to perform the transformations in the appropriate places in the logic.

Our script consists of four major steps: (1) obtaining an area and timing optimized technology mapped representation of each logic block, (2) determining a goal delay for each block corresponding to the minimum delay from primary inputs to primary outputs of the block (not considering inter-block inputs and outputs), (3) transforming the multi-phase network so as to satisfy the goal delay times as much as possible, and (4) assigning latches to inputs, outputs, and inter-block signals so as to ensure no race conditions and unclocked feedback paths.

The first step of the algorithm generates a starting point for the network. `misII` is used to generate an area and timing optimized multi-level logic implementation of each logic block [BRAY87]. The resulting logic is then technology mapped so that later operations can use an accurate delay model corresponding to the implementation technology.

The second step determines a *goal delay* for each block before moving any logic. The goal delay is defined as the minimum achievable delay from primary inputs to primary outputs of each logic block. It does not consider inputs from or outputs to

other logic blocks. The goal delay time for each block is obtained by running misII for timing optimization with primary input arrival times set to 0 while inter-block inputs are set to $-\infty$, inter-block output required times are set to $+\infty$, and minimizing the time for the primary outputs. This minimum time represents the best achievable input to output delay through the logic block by considering only the logic that must absolutely be within the block, that is, the logic to generate outputs at the end phase from inputs sampled at the start phase. It assumes that all other logic will not be on the critical path and if so that it can be moved to an earlier or later block. This is an ideal that may not necessarily achievable but serves as an upper bound on what is possible and is used to guide and/or limit the movement of logic between logic blocks. In section 6, we will measure the success of our algorithm based on how close the resulting circuit is to this idealized goal.

The third step of the algorithm applies a series of transformations to the network to attempt to satisfy the goal delays. These involve moving logic on the critical path backward or forward to another block and restructuring logic to enable such moves. The goal delay is used to focus where the transformations are applied. The script begins by not permitting moves that would violate the goal delay of the destination block. Later, the goal delays of the blocks are adjusted to enable moves which improve the total cycle time. Moves that would violate the goal delay of another block are permitted if they improve or do not have too detrimental an effect on the overall cycle time (a typical value is 20%). This capability lessens the greedy nature of the approach.

The last step assigns latches to the primary inputs and primary outputs of the logic as well as to enough inter-block signals to ensure that there will be no race conditions or unclocked feedback paths. The number of latches is minimized to reduce the area of the resulting circuit and their placement is optimized so as to have minimal impact on the final delays in the circuit.

Goal Delay

The goal delay for a block C_{ij} with start phase ϕ_i and end phase ϕ_j corresponds to the minimum separation between the rising edges of ϕ_i and ϕ_j. It is defined by the minimum time needed to generate the primary outputs of the block that are dependent on primary inputs and the latch time for those inputs. Time 0 corresponds to the rising edge of ϕ_i. The critical path in the logic then determines the position of the rising edge of ϕ_j.

The goal delay may also be user specified value if the phase is known to require some minimum duration due to constraints imposed by other parts of the design. In that case, the user supplied value is used and no goal delay is computed for the block.

An important assumption is being made in computing the goal delay. Namely, that state computation will not be on a critical cycle in the circuit and will therefore not limit the achievable cycle time. This assumptions is, of course, not always realistic. State computations can be quite extensive when state information is tightly encoded. Therefore, the goal delay is an objective or lower bound on the separation between clock phases. It simply provides information to direct the algorithm in selecting the blocks to which it should attempt to apply the transformations.

The goal delay of a block is computed by deleting all critical pre-computable logic. *Pre-computable logic* is the logic at the tail of a block C_{ij} that can be moved back. Logic can be moved back if all of its inputs arrive sufficiently early so that some computation can be achieved before the rising edge of ϕ_i. All logic in terms of primary inputs sampled on earlier phases than ϕ_i, outputs of other blocks with end phase ϕ_i, and inputs with sufficiently early arrival times with respect to ϕ_i may be pre-computable. To enable additional logic to be pre-computed the logic may need to be *restructured*. Restructuring is the process by which logic can be reorganized so that early arriving inputs are placed at the tail of the block. During goal delay computation only primary outputs are considered critical, state outputs have a required time of ∞. Therefore, all critical pre-computable logic will be in the transitive fanin of a primary output. Restructuring is accomplished using the misII delay optimization routines and information on how early the inputs are available relative to ϕ_i [SING88].

The amount of logic that can be moved into a block is constrained by the block's goal delay. Goal delays may be adjusted if increasing the delay of a block C_{ij} by Δ enables the delay of a block C_{jk} to decrease by more than Δ thus reducing the total cycle time, the goal for C_{ij} is adjusted upward by Δ and the goal for C_{jk} is adjusted downward to the delay of the new critical path of the block.

Based on user specified parameters, adjustments that increase the total cycle time may be allowed. This enables the optimization of blocks of

particular interest to the user and permits detrimental logic moves that lessen the greediness of the approach. Adjusting the goal upward when it appears unsatisfiable enables additional logic to be moved into the block, possibly decreasing the delay of other blocks. A downward adjustment may enable additional critical logic to be moved out of the block, thereby further reducing the total cycle time.

In summary, the goal delay is computed by repeatedly moving logic off the block, restructuring the logic to enable additional logic to be moved, and continuing this process until as much logic as possible has been moved. The goal delays may be adjusted to tradeoff critical paths in connected blocks.

Retiming the Multi-Phase Network

A multi-phase network is *retimed* by moving logic across latch boundaries. This is done so as to satisfy the goal delays of each logic block. The most important aspects of our retiming algorithm are: the heuristics used to determine the logic to move in each iteration, the restructuring performed in order to enable more moves, and the adjustments to goal delays.

Before attempting any moves a *slack* must be computed for each node in the combinational network of each logic block. An arrival time and a required time are computed for each input and output of a block. The required time for all outputs (primary outputs and state) is the block's goal delay which is propagated back to obtain a required time for each intermediate signal in the network. Arrival times corresponding to latch fanout delays are used for all inputs (primary inputs and state) and are propagated forward to obtain a corresponding arrival time for each signal. The slack for a signal is simply the difference between its required and arrival times. All signals within ε (same default value as in `misII`, .5) of the most negative slack are considered critical [SING88].

The delay of the critical path of a block is decreased by moving critical tail logic backward or critical head logic forward. To move a block of logic L off the tail of block C_{ij}, all inputs to L must be ready before ϕ_i by an amount of time equal to or greater than the delay of the logic in L. The block to which L is moved to must tolerate this additional logic without violating its goal delay. Similarly, logic that is not required to be valid at ϕ_j may be moved forward. The outputs of L (i.e., inputs to the next block) must have slack greater than or equal to the delay of L.

Logic is selected to be moved so as to have minimum impact on the critical paths of other blocks. Therefore, logic to be moved is prioritized based on the block to which it will be transferred as follows: (1) to pre-block or post-block logic that does not limit the overall cycle time, (2) to non-critical paths of other blocks, or (3) to other blocks that can tolerate an adjustment in their goal delay. Restructuring is used to create new logic to be moved. Also, the goal delays may be adjusted to enable additional moves as described above. The process of moving logic and adjusting goal delays continues until no beneficial transformations are possible.

IV. Latch Manipulations

Logic optimization is only part of the process of optimizing sequential logic. In the case of multi-phase sequential logic, the positioning of latches must also be considered for both timing and area optimization. This flexibility is available because latches are not required on every inter-block connection (see section 1).

Area and Timing Optimization

Our algorithm for positioning latches is based on a simple greedy algorithm that first enumerates all cycles in the multi-phase network and then *covers* each cycle with one or two latches depending on whether the latch type is edge-triggered or level-sensitive. By covering a cycle we mean placing latches at one or two inter-block connections on the cycle. The algorithm finds the most common inter-block connection and places a latch there. Any cycles that may become completely covered as a result of this placement are removed from the list. The process repeats until no cycles are left. Also, to prevent race conditions when using level sensitive latches there must be at least one latch (controlled by ϕ_j) along a path starting with a ϕ_i input and ending with ϕ_i output. These latch requirements are taken into account in the covering heuristic. Since a latch contributes to total area as much as a 2-input NAND gate (i.e., two literals), the total contribution can be significant. Our heuristic performs quite well consistently yielding solutions with less latches than manual designs.

Delay considerations are taken into account by weighting each inter-block connection point based on its slack. This favors placing latches where there is slack to absorb its delay. The value of the

weight is the minimum of the slack and latch delay values. Some latches may be initially specified as necessary if the input they drive has a large fanout. The removal of such a latch would have a negative effect on its input (the output of the previous block). Its removal is therefore prohibitive and its presence is taken into account in the positioning of the remainder of the latches.

Reset State Maintenance

Special care must be given in the movement of logic across latches when a reset state is specified for the finite-state machine. In many cases, the reset state is not reachable by any sequence of inputs but rather a separate reset signal is used to force the latches to the appropriate state. Our algorithm updates reset state information by keeping a list of these latches and updating it whenever a logic move is made across one of its elements. In some cases, this may mean the duplication of a latch if a logic move requires it to have both logic values at reset. The PODEM algorithm from test generation is used to determine the new reset values after a move. Reset state latches are marked for inclusion in the latch positioning step.

V. Implementation

Our multi-phase logic timing optimization algorithm is implemented as a set of extensions to the logic optimization tool `misII` (version 2.1) [BRAY87]. The extensions include support for the multi-phase structure of the circuit (i.e., separate but related logic blocks), capabilities for moving logic between blocks, and analysis and restructuring of block delays.

Support for Multi-Phase Networks

The input representation for the circuit is a single description in one of the standard input languages (e.g., blif) with extensions for specifying the phase during which each primary input will be sampled and the phase during which each primary output will be generated. All other signals are considered to be internal state. Furthermore, the specifications for the inputs (outputs) contain arrival (required) times as offsets before (after) the rising edge of the sampling (generating) phase.

Based on this added information, the single network is decomposed into separate networks for each of the combinational logic blocks in the circuit. Latches are not represented in the networks stored in `misII` but are added by a separate program called after the logic transformations. The logic network associated with each block may be area and timing optimized using any of the standard methods. For examples in this paper, we used the standard `misII` algebraic and Boolean area minimization scripts, the speed-up command, and technology mapping with delay optimizing parameters (i.e., *map -m .75*). A technology mapped `misII` network is maintained for each logic block of the multi-phase circuit.

Moving Logic Between Blocks

Routines have been implemented for moving logic forward and backward across logic blocks. These routines are used to move critical logic from the tail or head of a block, restructuring as necessary, until either the block has satisfied its goal delay or no movable logic can be found. Critical logic in block C_{ij} is identified and moved backward or forward to another block. Internally, this is accomplished by modifying the logic networks of the two blocks and updating timing delays for each node in the networks that are modified. The block from which logic was moved backward may have new inputs and the other block new outputs. Conversely, for the case of logic that is moved forward. This is currently implemented with signal renaming as logic is moved between blocks.

Moves that would violate the goal delay of the destination block are not permitted. The destination block must be able to absorb both the gate delay of the logic being moved and any fanout impact. Logic may be moved from the head (tail) of a block to the tail (head) of the same block.

The algorithm for generating and determining the logic to be moved back is similar to that used for computing the goal delay. Logic is not considered for moving unless all the inputs arrive at least one gate delay early. When there is no critical logic which can be moved, the block may be restructured in an attempt to obtain a new network with early arriving inputs at the tail of the block so that this may subsequently be moved. Non-critical logic may also be moved (particularly if it is in the transitive-fanin of the critical logic) as this may enable subsequent speed-up/restructuring that will permit other logic to move.

Timing Model and Restructuring

The timing model of the library used for technology mapping is used in all delay

calculations. A latch is assumed to be driving all inputs to a block and its drive capability is the same as that of a 2-input NAND gate. Of course, once latch optimization is completed, some of these latches may be removed if the inputs they drive can the absorb the additional fanout delay.

Restructuring is the process by which a new implementation of a critical path is obtained that moves late arriving inputs closer to the outputs and early arriving inputs closer to the tail of a block. Gates at the tail of the critical path can then be moved back to an earlier block. The process of generating a restructuring involves: (1) updating input arrival times, (2) calling the misII speed-up algorithm, (3) adjusting input delay times to reflect latch fanout delays, (4) technology mapping the new network, and (5) determining whether to accept the restructuring.

Restructuring in our algorithm relies on the `misII` *speed-up* command. However, due to the imprecise interaction between the 2-input gate based speed-up algorithm and technology mapping, as well as the dominating effect of inputs with very high drive, the restructuring is rejected if it does not enable moving the tail of the critical path off the block.

VI. Experimental Results

Our algorithm was used on a collection of finite-state machine examples to demonstrate the delay optimizations possible. However, only the SPUR CPU controller is a true multi-phase circuit [KONG89]. The others are finite-state machines from the benchmark suite developed for the International Workshop on Logic Synthesis [LISA88]. The results of timing optimization are presented by placing the output of our algorithm between two bounds. The upper bound (or worst case) delay is that of the starting point for the algorithm, namely, the output of `misII` area, timing, and technology mapping optimizations. This represents what is currently available to designers. The lower bound (or best case) delay is based on the idealized goal delays for each logic block. The success of the algorithm is measured by how close the result is to the goal delays. Recall that the goal delay may not be realistic.

Our timing optimization algorithm utilizes many parts of the `misII` logic optimization tool. Its running time on the results presented in this section ranges from 2.5 to 9.0 times the time required to use `misII` to obtain the initial optimized and mapped network (4.5 times on average).

The SPUR CPU Controller

The SPUR CPU controller is an example of a four-phase sequential circuit. It was designed as two separate finite-state machines each having inputs and outputs on all four clock phases. The structure of the circuit is illustrated in Figure 4. Basically, the state computation logic is concentrated during one phase and output logic is distributed so as to be as close as possible to the outputs.

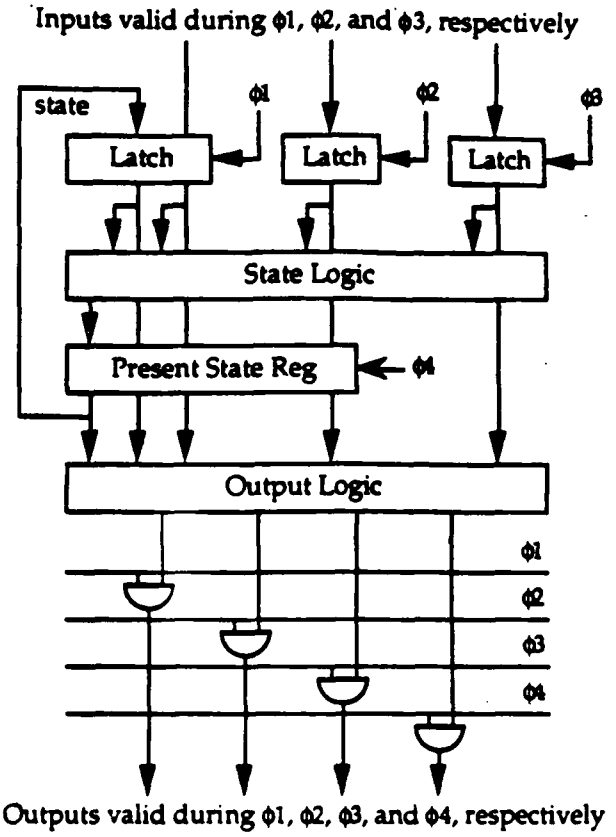

Figure 4

Our objective was to obtain a faster cycle time by precomputing logic and distributing state logic across all phases. The results are shown in Table 2. Since cycle time was the primary criteria for optimization, no detrimental moves were accepted. Two encodings for the states were used. The first corresponds to the actual SPUR encoding and the second to a one-hot encoding scheme.

	MIS	Opt	Goal	Impr%	Goal%
SPUR Encoded					
C1 time	6.20	5.40	4.50	12.9	47.1
C2 time	11.30	8.10	7.50	28.3	84.2
C3 time	11.10	8.80	8.70	20.7	95.8
C4 time	5.90	4.90	5.80	17.0	-
cycle time	34.50	27.20	26.50	21.2	91.2
literals	323	375		-16.1	
latches	62	83		-33.9	
total area	447	541		-21.0	
SPUR 1-hot					
C1 time	7.30	5.40	4.50	26.0	67.9
C2 time	8.90	7.70	6.80	13.5	57.1
C3 time	8.90	8.30	8.90	6.7	-
C4 time	5.20	5.60	5.10	-7.7	-400.0
cycle time	30.30	27.00	25.30	10.9	66.0
literals	287	301		-4.9	
latches	79	79		0.0	
total area	445	459		-3.2	

Table 2

The retiming optimizations resulted in a network where the phases were more balanced in terms of logic delays and the overall cycle time was faster. The retimed one-hot representation was 10.9% faster than the starting point (i.e., the fastest network obtainable using the delay optimization capabilities in `misII`) with a 3.2% area penalty. The encoded representation was 21.2% faster with a 21.0% area penalty. These results are 66.0% and 91.2%, respectively, of the difference between the delays achievable with `misII` and the idealized goal times. (The result of optimization is occasionally better than the goal due to the different networks generated during retiming and the different optimizations they enable.)

Logic Synthesis Benchmarks

The International Workshop on Logic Synthesis benchmarks are a collection of single-phase finite-state machines. We viewed them as two-phase circuits with inputs arriving on one phase and outputs being generated on the other. This would be a common implementation style in MOS technology. Our objective was to minimize input to output delay (or just one phase's logic) rather than overall cycle time. The reduction in output delay can be valuable in generating control signals for other parts of the circuits and allowing for wiring delays. The script was written to accept moves that would permit up to a 20% increase in cycle time. Table 3 shows the average results for 17 of the benchmarks (bbara, bbtas, cse, dk14, dk16, dk17, dk512, ex1, ex3, ex6, ex7, keyb, lion9, mark1, sand, s1, styr).

	C1 Impr%	Area Penalty	Cycle Impr%	C1 Goal%
Enc-Alg	16.3	7.5	-2.3	64.0
Enc-Bool	30.2	19.3	6.7	81.6
1Hot-Alg	11.3	8.4	5.3	58.0
1Hot-Bool	9.3	8.7	3.9	52.7
Average	19.9	11.4	3.7	71.6

Table 3

Several variants of the examples were used to collect the information presented in Table 3. The *Enc* examples were encoded using the fanin-oriented option of `mustang` [DEVA88]. The *1Hot* examples used a one-hot state encoding. *Alg* or *Bool* indicates whether the standard algebraic or standard Boolean `misII` script was used.

On the average, our retiming optimizations enable an input to output delay improvement of 19.9% with an average area penalty of 11.4%. This improvement is 71.6% of the difference between what `misII` alone can achieve and the idealized goal times. Changes in the total cycle time are generally small (i.e. 3.7% better on average), although some circuits did increase the cycle time close to the 20% allowed.

One last point to be made is that the comparisons of Table 3 assume an initial C_{12} goal time of 0. This is never the case. In reality, a second clock phase is used for precharging control and/or for other computation. In the case of precharging, it will be 10-25% of the duration of the main computational clock phase. In the case of computation during both phases, they will usually be within 50% of each other in duration and frequently comparable. In these cases, our algorithm is even more useful in that logic is distributed across both phases and all available time is utilized. These are the typical situations in which we expect our tool to be used.

VII. Summary and Conclusions

Timing optimization of multi-phase sequential logic is important in the design of high-performance MOS circuits. We have described an algorithm that yields substantial improvements on a variety of test cases. Input-output delays on a set of two-

phase benchmarks have been reduced by 20% to within 72% of the goal delays. Cycle time for a four-phase example was reduced by 21% to within 91% of the goal delays. Area penalties are of the same order and usually much less than the timing improvements. The algorithm is implemented as a script exploiting a set of extensions to the logic optimization tool `misII`.

Many improvements to our algorithm and implementation are possible. Three are immediately obvious. First, additional delay optimizations may be possible by exploiting the active time of the dynamic for computation [DAGE89]. Second, area penalties can be reduced by exploiting interblock don't care information and considering trading delay for area when performing the retiming [BART88]. Third, transistor sizing algorithms can make many retiming moves unnecessary for custom technologies and these should be made only if they also reduce area.

Acknowledgements

We are grateful to Carl Ebeling for initially motivating this research with the routing switch example and to Shing Ip Kong and Daebum Lee for providing an excellent test case from the SPUR CPU controller design. Discussions with Robert Brayton, Giovanni De Micheli, Sharad Malik, and Ellen Sentovich helped to clarify many of our ideas. Finally, the work presented in this paper would not have been possible without the excellent VLSI design and CAD environment provided by the Northwest Laboratory for Integrated Systems in the Department of Computer Science at the University of Washington. The NW-LIS and this research are supported by a grant from the Defense Advance Research Projects Agency under contract #N00014-88-K-0453.

References

[BART87] K. Bartlett, et. al., "BOLD: A Multiple-Level Logic Optimization System", *Proc. Int'l Conf. on Computer Aided Design* (Santa Clara), Nov. 1987.

[BART88] K. Bartlett, et. al., "Multilevel Logic Minimization Using Implicit Don't Cares", *IEEE Transactions on CAD*, Vol CAD-7, No. 6, June 1988.

[BRAY87] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang, "MIS: A Multiple-Level Logic Optimization System", *IEEE Transactions on CAD*, Vol CAD-6, No. 6, November 1987, pp. 1062-1081.

[DAGE89] M. Dagenais and N. Rumin, "On the Calculation of Optimal Clocking Parameters in Synchronous Circuits with Level-Sensitive Latches," *IEEE Transactions on CAD*, Vol CAD-8, No. 3, March 1989, pp. 268-278.

[DEMI89] G. De Micheli, "Synchronous Logic Synthesis", *Proceedings of International Workshop on Logic Synthesis*, (RTP, NC) May 1989.

[DEVA88] S. Devadas, H. Ma, A.R. Newton, and A. Sangiovanni-Vincentelli. "Mustang: State Assignment of Finite State Machines Targeting Multi-Level Implementations. *IEEE Transactions on CAD*, Vol CAD-7, No. 12, December 1988, pp. 1290-1300.

[EBEL88] C. Ebeling, private communication, 1988.

[HILL86] Mark Hill, et. al., "Design Decisions in Spur", *Computer*, November 1986, pp. 8-22.

[KONG89] S. Kong, "The SPUR CPU Behavioral Model" Report No. UCB/CSD 89/508, Computer Science Division (EECS) University of California, Berkeley, May 1989.

[LEIS83] C. Leiserson, F. Rose, and J. Saxe, "Optimizing Synchronous Circuitry by Retiming," *Proc. of the 3rd Caltech Conference on VLSI*, March 1983.

[LISA88] R. Lisanke, "Logic Synthesis and Optimization Benchmarks User Guide Version 2.0", Technical Report, Microelectronics Center of North Carolina, December 1988.

[MALI89] S. Malik, E. Sentovich, R. Brayton, and A. Sangiovanni-Vincentelli, "Retiming and Resynthesis: Optimizing Sequential Networks with Combinational Techniques", *Proc.of Int'l Workshop on Logic Synthesis*, (RTP, NC) May 1989.

[SAXE85] J. Saxe, "Decomposable Searching Problems and Circuit Optimization by Retiming: Two Studies in General Transformations of Computational Structures", Ph.D Dissertation, Dep't of Computer Science, Carnegie Mellon University, 1985.

[SING88] K. Singh, A. Wang, R. Brayton, and A. Sangiovanni-Vincentelli,"Timing Optimization of Combinational Logic", *Proc. Int'l Conf. on Computer Aided Design* (Santa Clara), Nov. 1988, pp. 282-285.